King Mongkut's University of Technology Thonburi

Graphs

CSC 209 Data Structures

Dr. Kulwadee Somboonviwat

School of Information Technology, KMUTT

kulwadee.som [at] sit.kmutt.ac.th

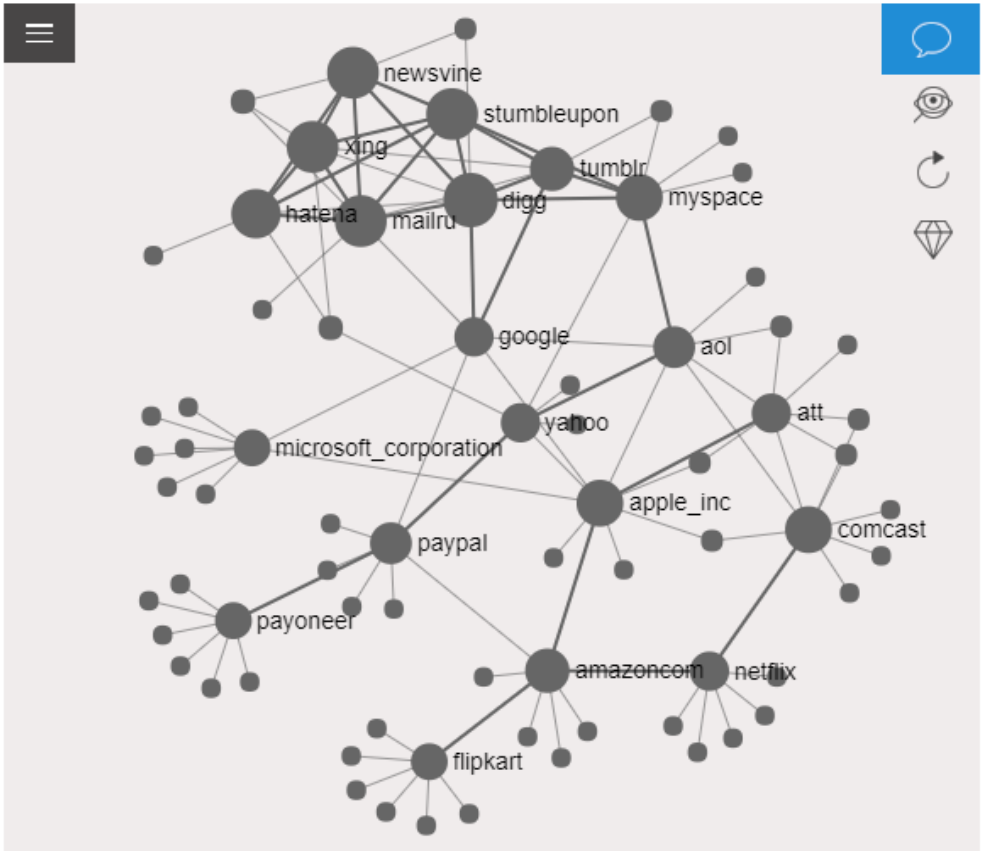
Lecture Plan

- Graph definition
- Graph representation
  - Adjacency matrix
  - Adjacency list
  - Graph API
- Depth first search
  - Case study: Knight’s Tour Problem
- Breadth first search
  - Case study: Shortest path on unweighted graph

Graph

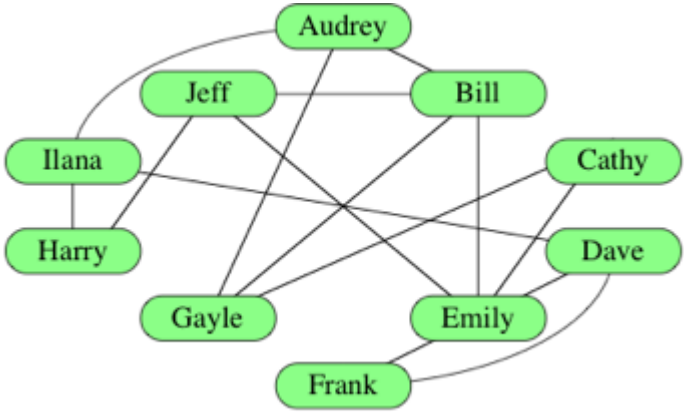
- Set of **vertices** connected pairwise by **edges**.

graph	vertex	edge
communication	telephone, computer	fiber optic cable
circuit	gate, register, processor	wire
mechanical	joint	rod, beam, spring
financial	stock, currency	transactions
transportation	intersection	street
internet	class C network	connection
game	board position	legal move
social relationship	person	friendship
neural network	neuron	synapse
protein network	protein	protein-protein interaction
molecule	atom	bond

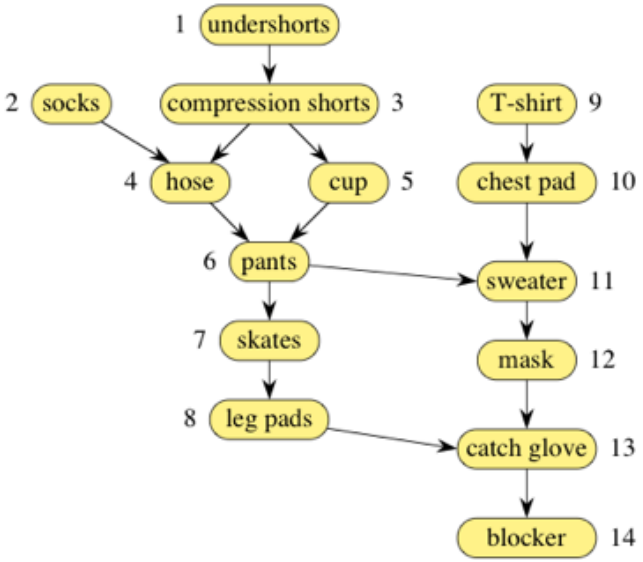


http://noduslabs.com/portfolio/hitech-companies/

This is a network graph of the main hitech companies and their relations to one another. The data was manually derived from Google Knowledge Graph, so it reflects which companies people search for together.



Social relationship graph

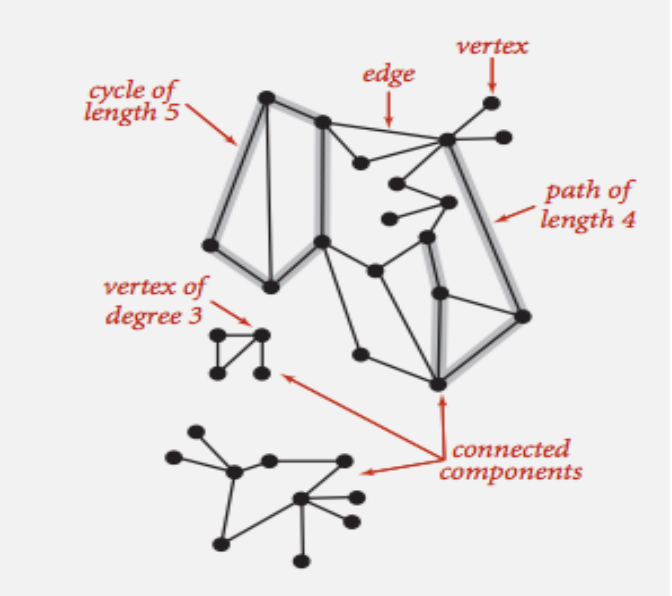


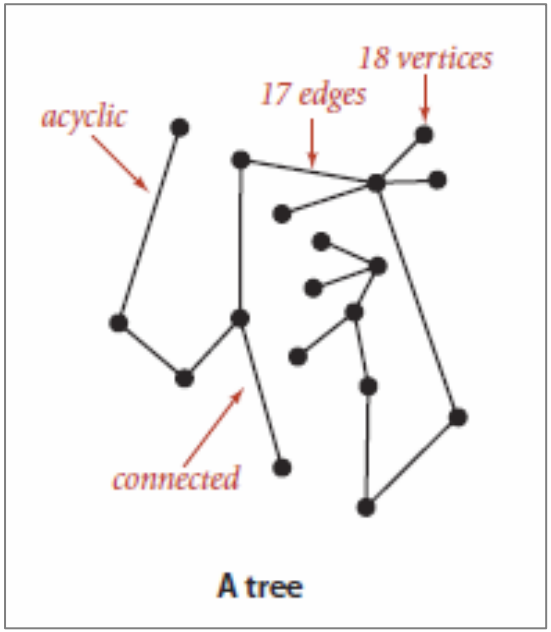
Order of wearing an ice hockey dress

https://www.khanacademy.org/computing/computer-science/algorithms/graph-representation/a/describing-graphs

Graph terminology

- *Path*. Sequence of vertices connected by edges.
- *Cycle*. Path whose first and last vertices are the same.
- Two vertices are *connected* if there is a path between them.
- When two vertices *v* and *w* are directly connected with an edge, we say *v* and *w* are *adjacent* to one another and that the edge is *incident* to both vertices.





- A *tree* is an acyclic connected graph. A disjoint set of trees is called a *forest*.

Some graph-processing problems

- **Path**. Is there a path between vertex *s* and vertex *t* ?
- **Shortest path**. What is the shortest path between vertex *s* and vertex *t* ?

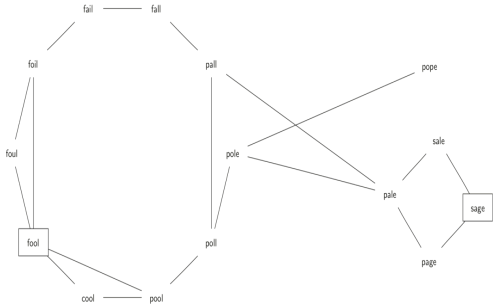
- **Cycle**. Is there a cycle in the graph?
- **Euler tour**. Is there a cycle that uses each edge exactly once?
- **Hamilton tour**. Is there a cycle that uses each vertex exactly once?

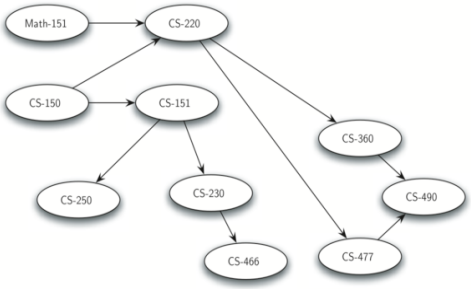
- **Connectivity**. Is there a way to connect all of the vertices?
- **Minimum Spanning Tree**. What is the best way to connect all the vertices?
- **Bi-connectivity**. Is there a vertex whose removal disconnects the graph?

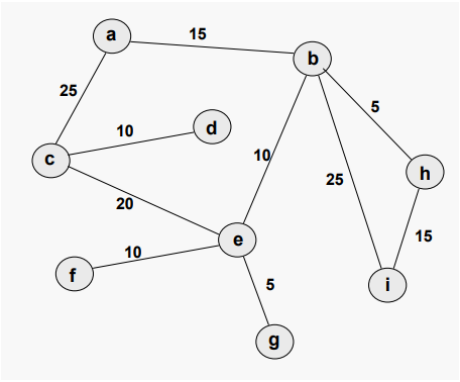
- **Planarity**. Can you draw the graph in the plane with no crossing edges?
- **Graph isomorphism**. Do two adjacency lists represent the same graph?

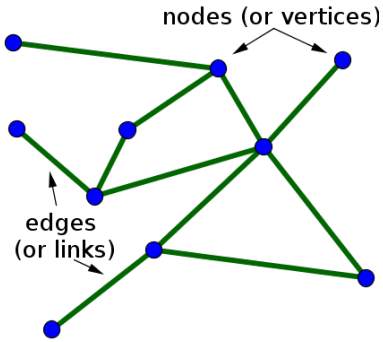
Challenge. Which of these problems are easy? Difficult? Intractable?

Graph Models

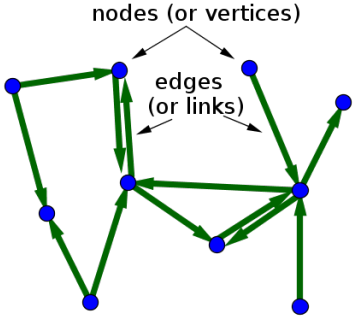




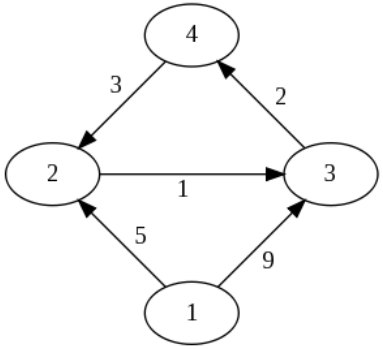




Undirected graph – all edges are bidirectional



Directed graph (or digraph) – all edges are directed from one node to another.

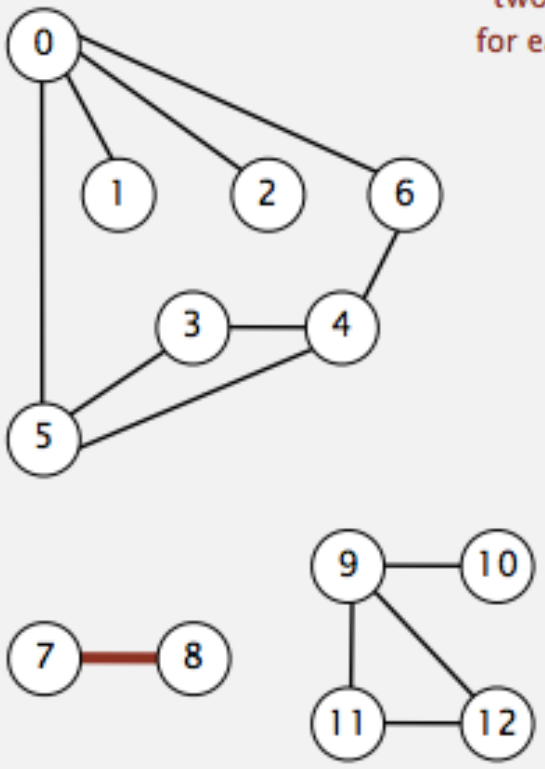


Edge Weighted Graph – all edges are associated with numerical values, called weights

Lecture Plan

- Graph definition
- Graph representation
  - Adjacency matrix
  - Adjacency list
  - Graph API
- Depth first search
  - Case study: Knight’s Tour Problem
- Breadth first search
  - Case study: Shortest path on unweighted graph

Adjacency Matrix Graph Representation

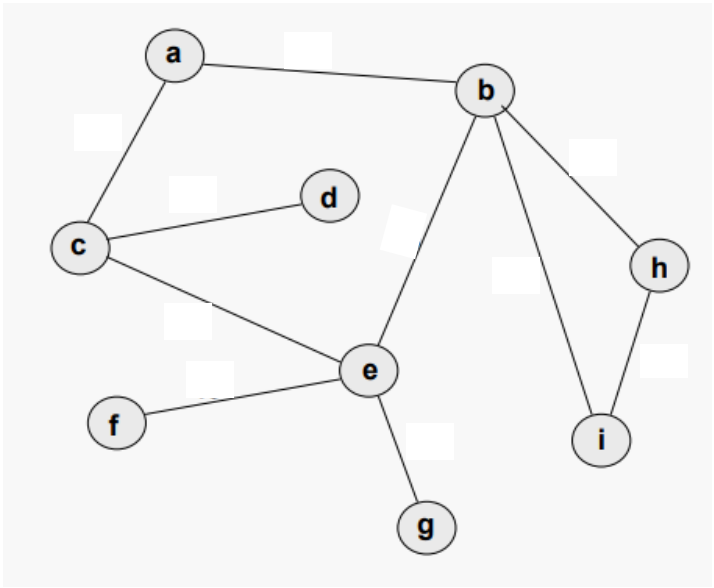


two entries
for each edge

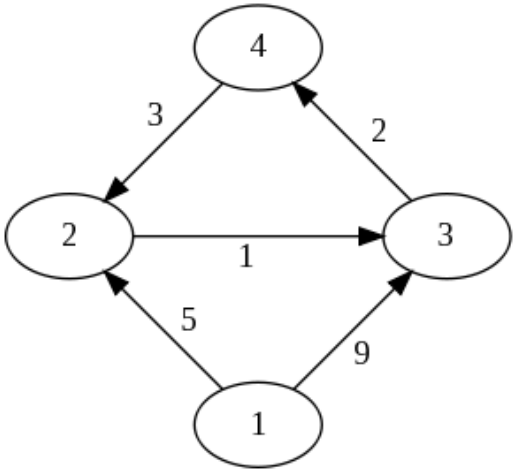
	0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	0	0	1	1	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0	0	0	0
4	0	0	0	1	0	1	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0	0	0	0	0	0	0
6	1	0	0	0	1	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	1	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	1	1	1
10	0	0	0	0	0	0	0	0	0	1	0	0	0
11	0	0	0	0	0	0	0	0	0	1	0	0	1
12	0	0	0	0	0	0	0	0	0	1	0	1	0

Q. How long to iterate over vertices adjacent to v ?

Exercise. Represent the following graph using an adjacency matrix

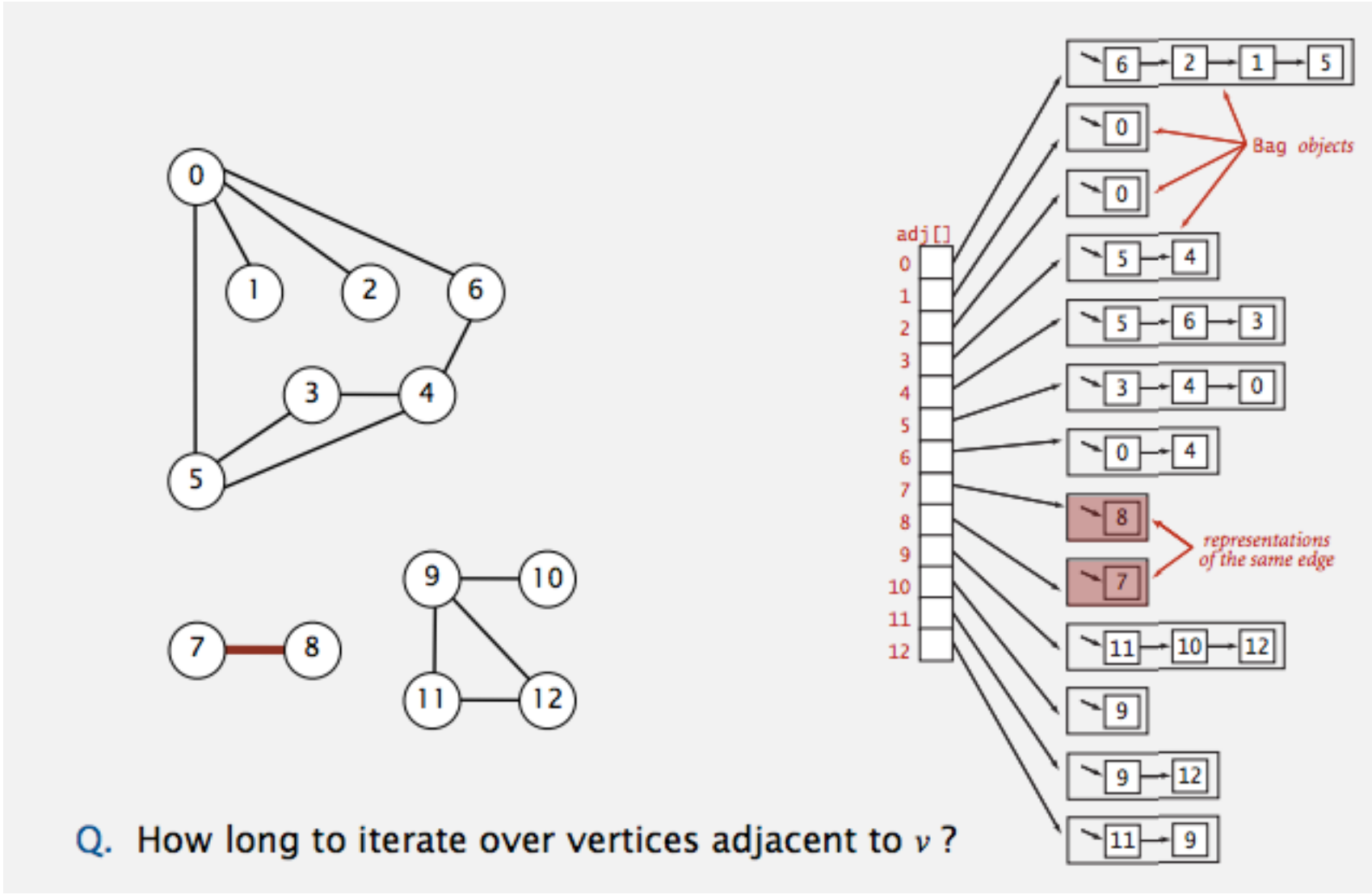


(a)

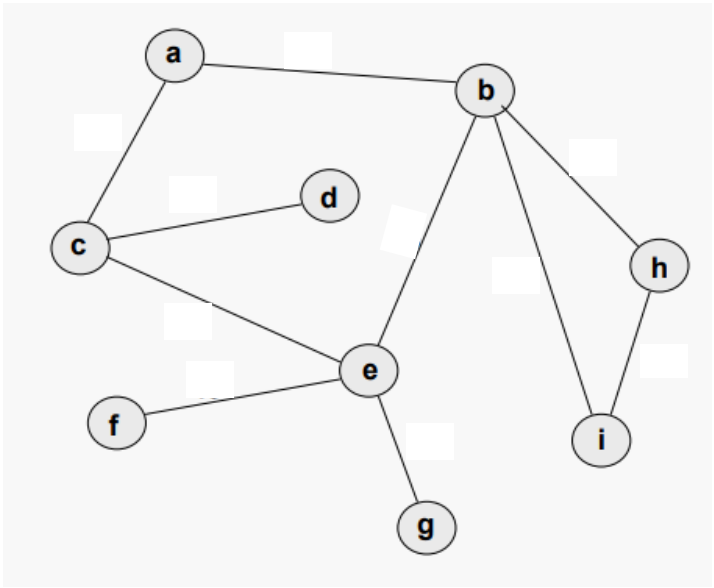


(b)

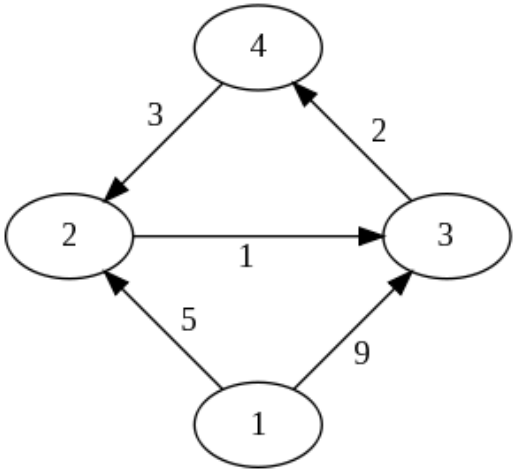
Adjacency List Graph Representation



Exercise. Represent the following graph using an adjacency list



(a)

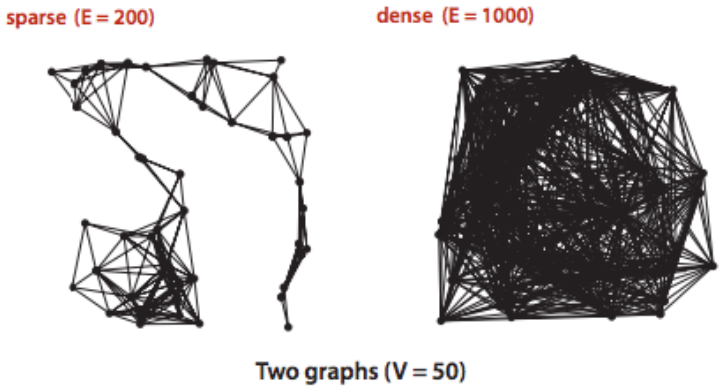


(b)

Graph representation

In practice. Use adjacency list representation

- Algorithms based on iterating over vertices adjacent to v.
- Real-world graphs tend to be **sparse.**



representation	space	add edge	edge between v and w?	iterate over vertices adjacent to v?
list of edges	E	1	E	E
adjacency matrix	V^2	1 *	1	V
adjacency lists	$E + V$	1	*degree*(v)	*degree*(v)

Graph API

public class Graph		
	Graph(int V)	*create an empty graph with V vertices*
	Graph(In in)	*create a graph from input stream*
void	addEdge(int v, int w)	*add an edge v-w*
Iterable<Integer>	adj(int v)	*vertices adjacent to v*
int	V()	*number of vertices*
int	E()	*number of edges*

Graph Java Implementation

```
public class Graph
{
   private final int V;
   private Bag<Integer>[] adj;

   public Graph(int V)
   {
      this.V = V;
      adj = (Bag<Integer>[]) new Bag[V];
      for (int v = 0; v < V; v++)
         adj[v] = new Bag<Integer>();
   }

   public void addEdge(int v, int w)
   {
      adj[v].add(w);
      adj[w].add(v);
   }

   public Iterable<Integer> adj(int v)
   {  return adj[v];  }
}
```

adjacency lists
(using Bag data type)

create empty graph
with V vertices

add edge v-w
(parallel edges and
self-loops allowed)

iterator for vertices adjacent to v

Bag

public class Bag<Item> implements Iterable<Item>		
	Bag()	*create an empty bag*
void	add(Item item)	*add an item*
boolean	isEmpty()	*is the bag empty?*
int	size()	*number of items in the bag*

```
import java.util.Iterator;

public class Bag<Item> implements Iterable<Item>
{
   private Node first;  // first node in list

   private class Node
   {
       Item item;
       Node next;
   }

   public void add(Item item)
   {  // same as push() in Stack
      Node oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public Iterator<Item> iterator()
   {  return new ListIterator();  }

   private class ListIterator implements Iterator<Item>
   {
       private Node current = first;

       public boolean hasNext()
       {  return current != null;  }

       public void remove() { }

       public Item next()
       {
           Item item = current.item;
           current = current.next;
           return item;
       }
   }
}
```

This `Bag` implementation maintains a linked list of the items provided in calls to `add()`. Code for `isEmpty()` and `size()` is the same as in `Stack` and is omitted. The iterator traverses the list, maintaining the current node in `current`. We can make `Stack` and `Queue` iterable by adding the code highlighted in red to ALGORITHMS 1.2 and ALGORITHM 1.3, because they use the same underlying data structure and `Stack` and `Queue` maintain the list in LIFO and FIFO order, respectively.

Lecture Plan

- Graph definition
- Graph representation
  - Adjacency matrix
  - Adjacency list
  - Graph API
- Depth first search
  - Case study: Knight’s Tour Problem
- Breadth first search
  - Case study: Shortest path on unweighted graph

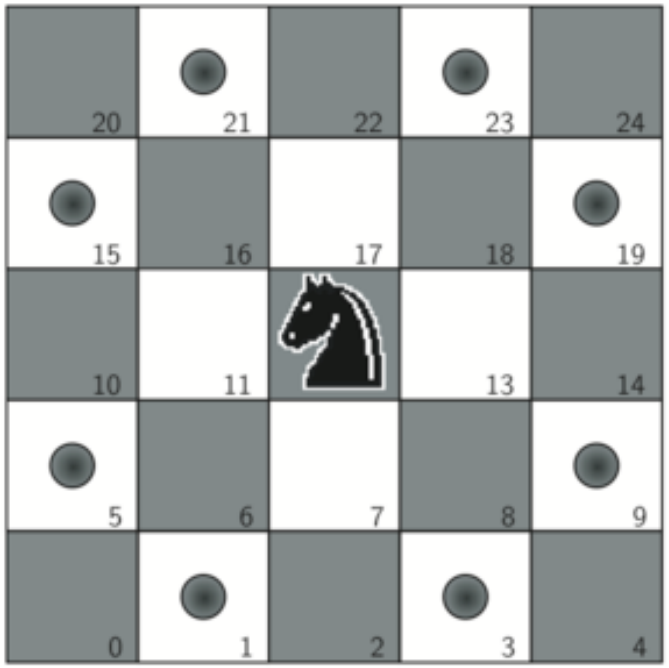
Knight's Tour Problem

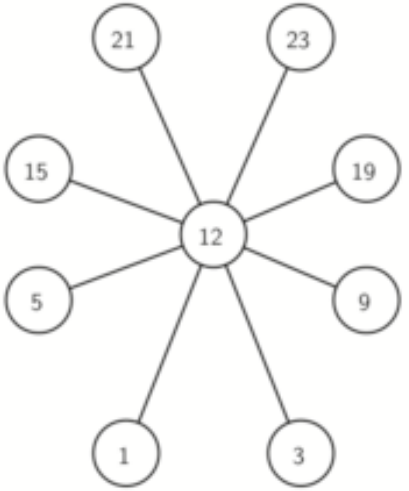
- Played on a chess board with a single chess piece, the **knight**
- Goal. Find a sequence of moves that allow the knight to visit every square on the board exactly once.
- **Solution.**
  - Represent the legal moves of a knight on a chessboard as a graph
  - Use a *depth first search* algorithm to *find a path of length rows x columns-1* where every vertex on the graph is visited exactly once.

https://en.wikipedia.org/wiki/Knight_(chess)

Knight's Tour Problem

- Knight can move across the board in either 2 squares horizontally and 1 square vertically, or 1 square horizontally and 2 squares vertically

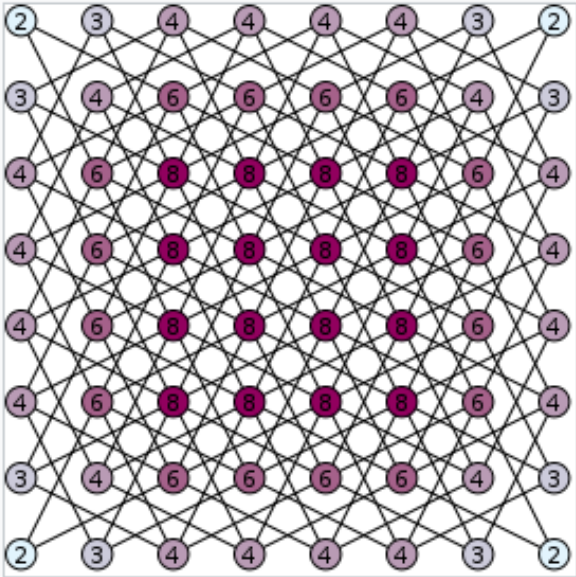




http://interactivepython.org/runestone/static/pythonds/Graphs/BuildingtheKnightsTourGraph.html

Knight's Tour Problem

The graph of all possible moves on an 8x8 board.
(V = 64 nodes, E = 336 edges; edge density = 336/(64*63)/2))

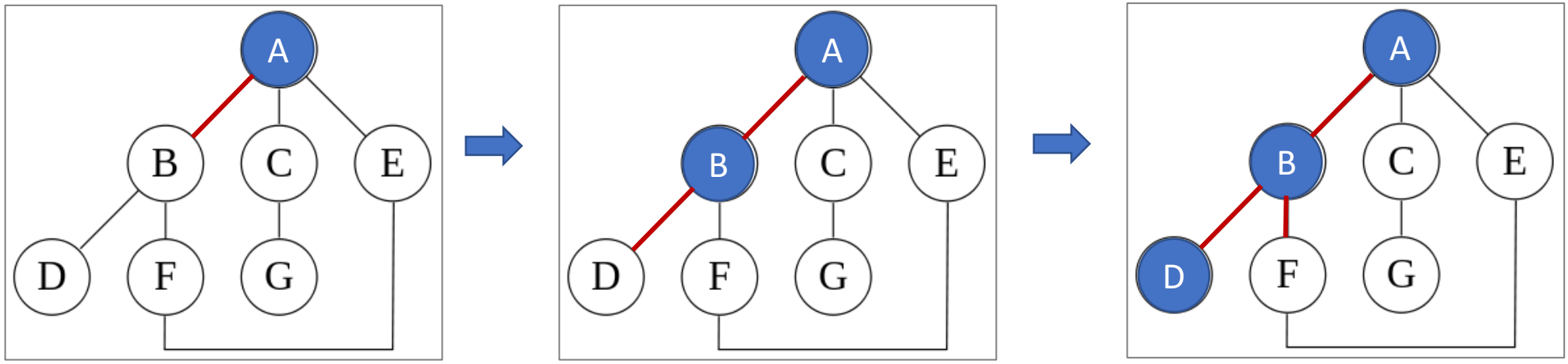


Knight's graph showing all possible paths for a knight's tour on a standard 8×8 chessboard. The numbers on each node indicate the number of possible moves that can be made from that position.

http://interactivepython.org/runestone/static/pythonds/Graphs/BuildingtheKnightsTourGraph.html
https://en.wikipedia.org/wiki/Knight%27s_tour

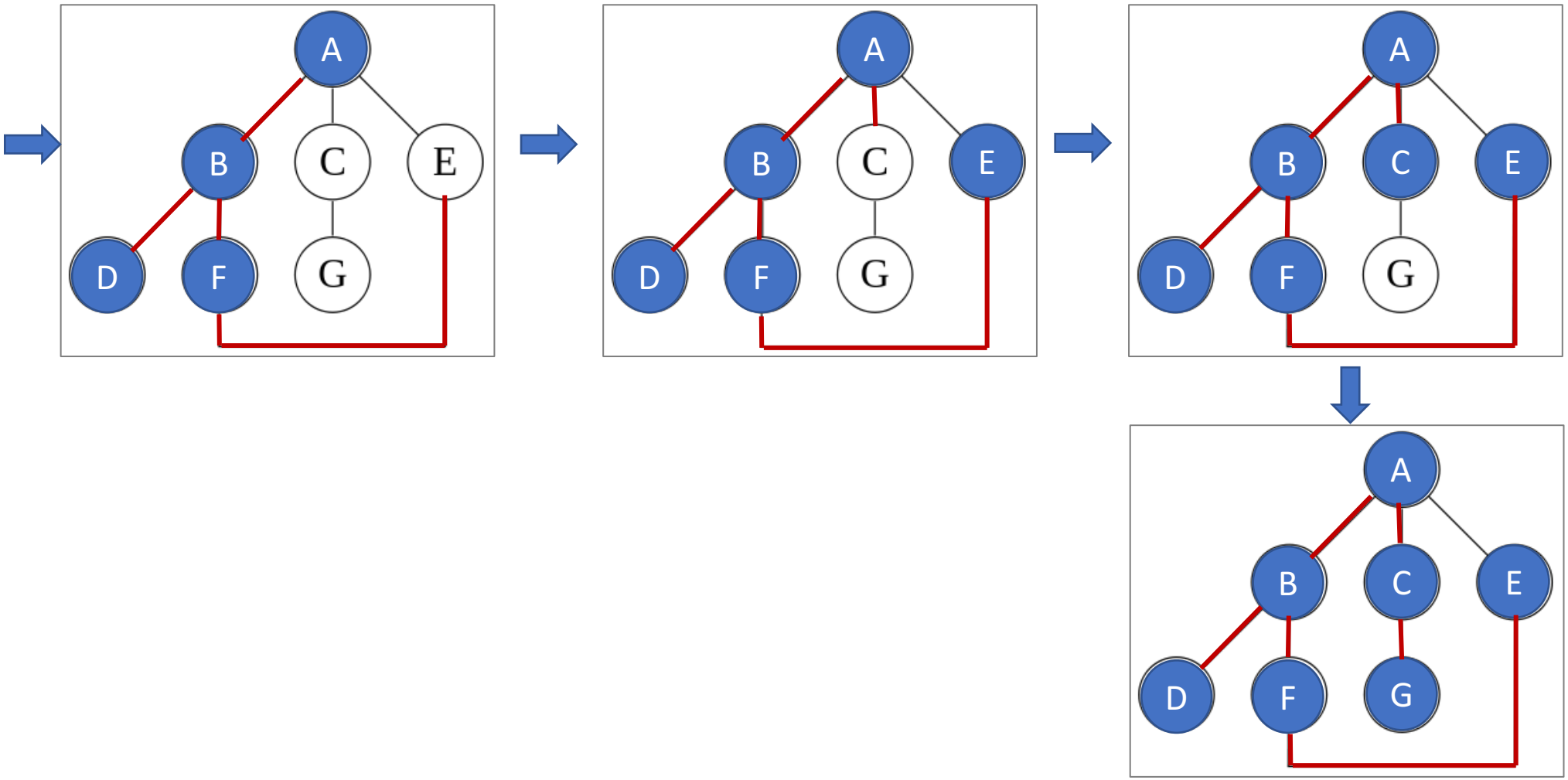
Depth First Search Algorithm

dfs(G, A)



Depth First Search Algorithm

dfs(G, A)



Depth First Search Algorithm

Running time :
O(V + E)

Input: A graph *G* and a vertex *v* of G

Output: All vertices reachable from *v* labeled as discovered

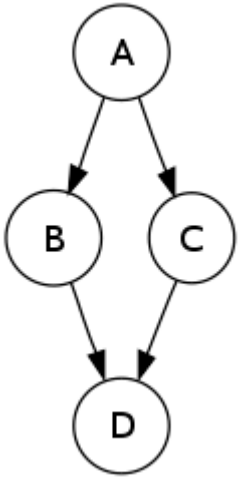
A recursive implementation of DFS:[5]

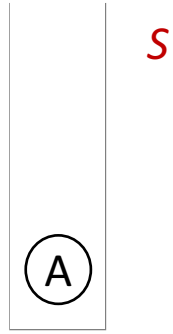
```
1  procedure DFS(G,v):
2      label v as discovered
3      for all edges from v to w in G.adjacentEdges(v) do
4          if vertex w is not labeled as discovered then
5              recursively call DFS(G,w)
```

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Depth First Search Algorithm

dfs(G, A)



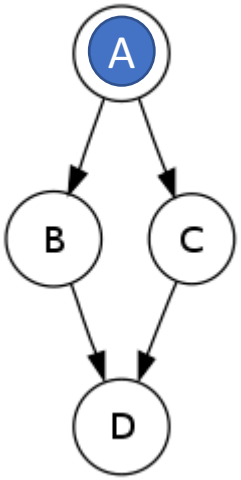


S

```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Depth First Search Algorithm

dfs(G, A)



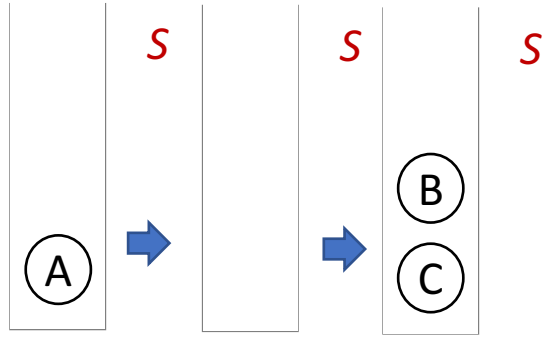
v = (A)

Marked A as discovered

// G.adjacent(A) = {'B', 'C'}

for each node in G.adjacent(A):

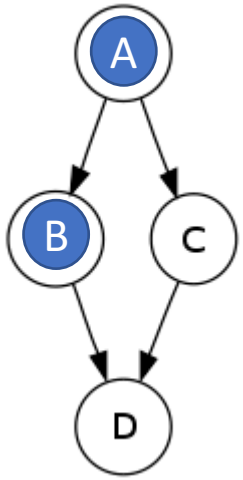
S.push(node)




```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Depth First Search Algorithm

dfs(G, A)



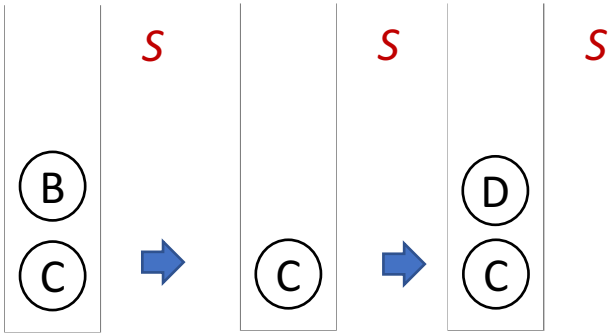
v = (B)

Marked B as discovered

// G.adjacent(B) = {'D'}

for each node in G.adjacent(B):

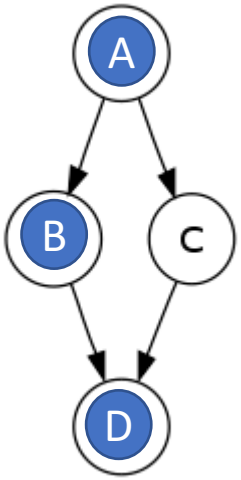
    S.push(node)




```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Depth First Search Algorithm

dfs(G, A)



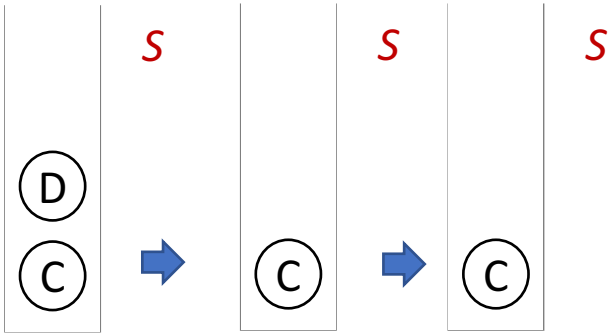
v = (D)

Marked D as discovered

// G.adjacent(D) = { }

for each node in G.adjacent(D):

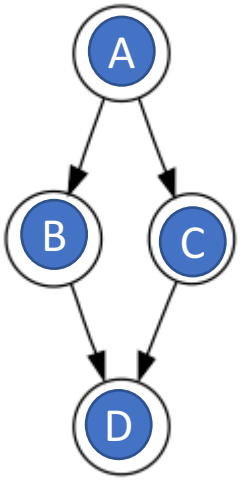
S.push(node)




```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

Depth First Search Algorithm

dfs(G, A)



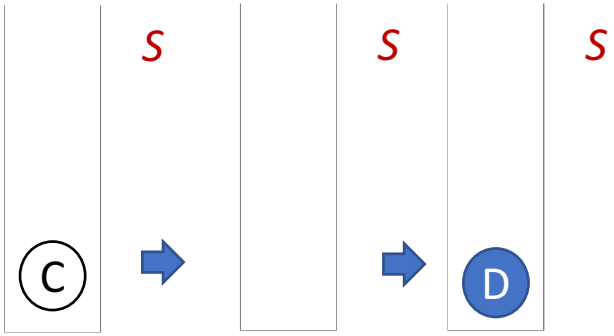
v = (C)

Marked C as discovered

// G.adjacent(C) = { 'D' }

for each node in G.adjacent(C):

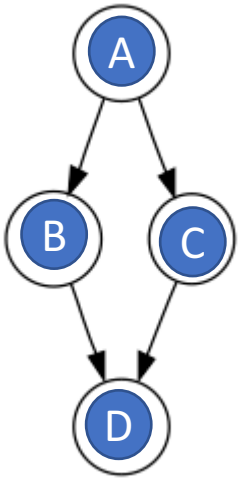
S.push(node)




```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

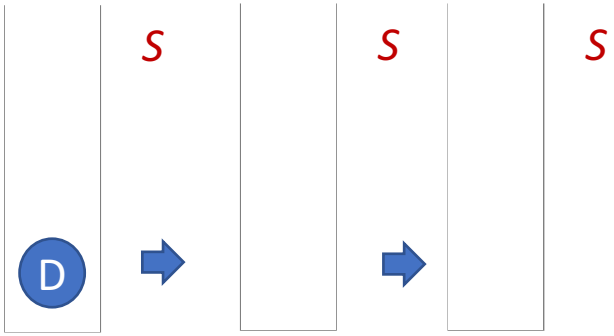
Depth First Search Algorithm

dfs(G, A)



v = D

// D is already discovered




```
1  procedure DFS-iterative(G,v):
2      let S be a stack
3      S.push(v)
4      while S is not empty
5          v = S.pop()
6          if v is not labeled as discovered:
7              label v as discovered
8              for all edges from v to w in G.adjacentEdges(v) do
9                  S.push(w)
```

DFS node ordering:

A B D C

Depth first search's Applications

- Knight's tour problem
- Solving maze puzzle / maze generation
- Finding connected component
- Topological sorting
- Web crawling

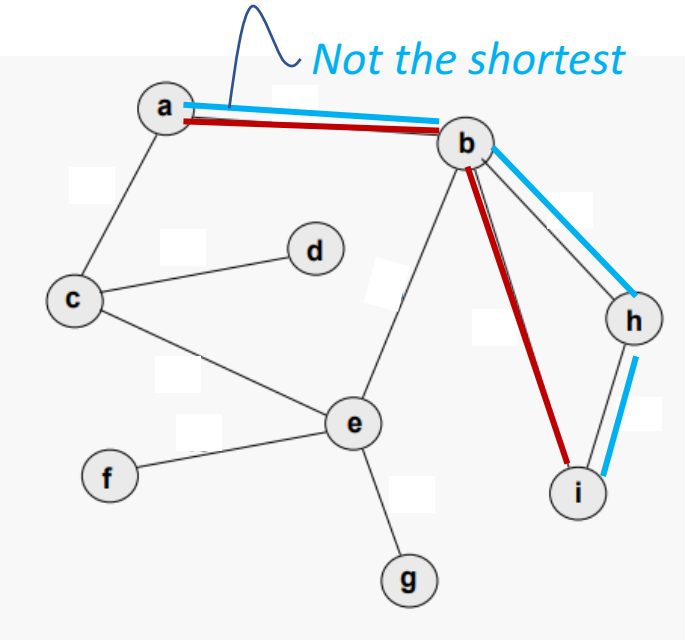
Lecture Plan

- Graph definition
- Graph representation
  - Adjacency matrix
  - Adjacency list
  - Graph API
- Depth first search
  - Case study: Knight’s Tour Problem
- Breadth first search
  - Case study: Shortest path on unweighted graph

Unweighted Shortest Path Problem

Input. A graph *G* and a source vertex *s* in *G*.

Output. Shortest paths from *s* to all vertices in *G*



Solution. Breadth first search from node *s*

a to b : a - b
a to c : a - c
a to d : a - c - d
a to e : a - c - e
a to f : a - c - e - f
a to g : a - c - e - g
a to h : a - b - h
a to i : a - b - i

Breadth first search algorithm

Input. A graph *G* and a source vertex *s* in *G*.

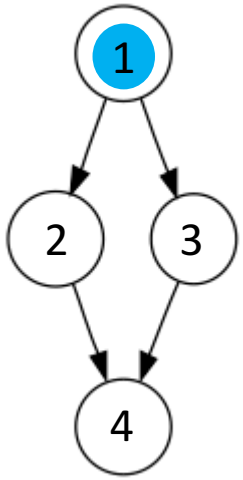
Output. Shortest paths from *s* to all vertices in *G*

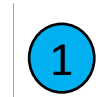
Running time :
O(V + E)

```
Breadth-first-search(G, s):
        int[] distance = new int[G.V()];
        int[] previous = new int[G.V()];
        Queue Q = new Queue();
        distance[s] = 0;
        previous[s] = 0;    // no previous node because s is the source
        Q.enqueue(s);
        mark s as discovered
        while Q is not empty do:
                v = Q.dequeue();
                for each vertex w adjacent to v do:
                        if w is not labeled as discovered then:
                                mark w as discovered
                                distance[w] = distance[v] + 1;
                                previous[w] = v;
                                Q.enqueue(w);
```

Breadth First Search Algorithm

bfs(G, 1)





Q

distance

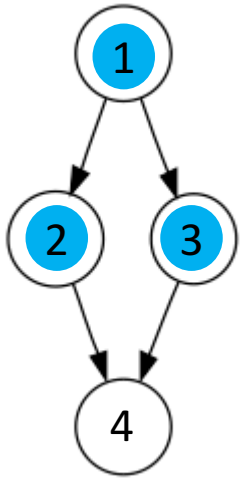
1	2	3	4
0			

previous

1	2	3	4
0			

Breadth First Search Algorithm

bfs(G, 1)



v = 1

// G.adj(v) = {2,3}

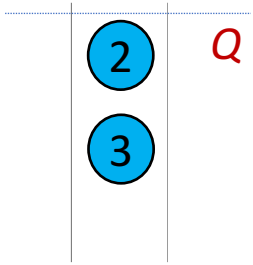
for each undiscovered node in G.adj(v):

    mark the node as discovered

    Q.enqueue(node)

    distance[node] = distance[v]+1

    previous[node] = v



distance

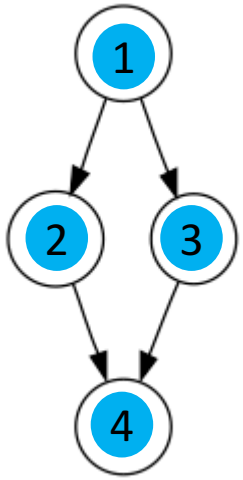
1	2	3	4
0	1	1	

previous

1	2	3	4
0	1	1	

Breadth First Search Algorithm

bfs(G, 1)



v = 2

// G.adj(v) = {4}

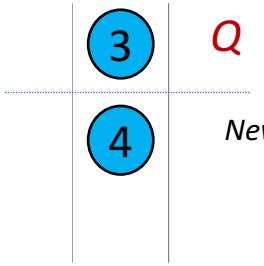
for each undiscovered node in G.adj(v):

    mark the node as discovered

    Q.enqueue(node)

    distance[node] = distance[v]+1

    previous[node] = v



distance

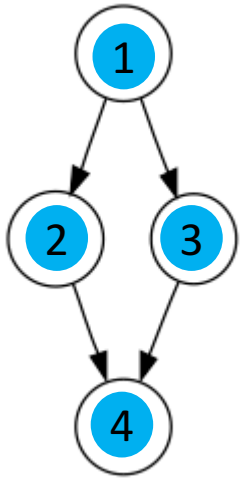
1	2	3	4
0	1	1	2

previous

1	2	3	4
0	1	1	2

Breadth First Search Algorithm

bfs(G, 1)



v = 3

// G.adj(v) = {4}

for each undiscovered node in G.adj(v):

    mark the node as discovered

    Q.enqueue(node)

    distance[node] = distance[v]+1

    previous[node] = v

4 *Q*

distance

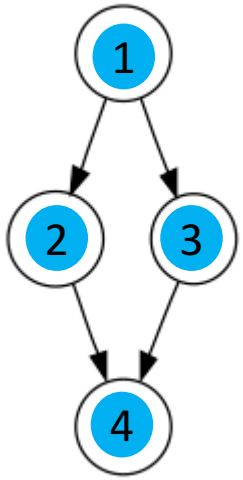
1	2	3	4
0	1	1	2

previous

1	2	3	4
0	1	1	2

Breadth First Search Algorithm

bfs(G, 1)



v = 4

// G.adj(v) = { }

for each undiscovered node in G.adj(v):

    mark the node as discovered

    Q.enqueue(node)

    distance[node] = distance[v]+1

    previous[node] = v

Q

distance

1	2	3	4
0	1	1	2

previous

1	2	3	4
0	1	1	2

[Output] Shortest path from:

1 to 2 : 1 - 2

1 to 3 : 1 - 3

1 to 4 : 1 - 2 - 4

Breadth first search's Applications

- Word ladder game
- Shortest path finding on unweighted graph
- Garbage collection
- String pattern matcher
- Computing maximum flow
- ….